Concurrency Control - Formal Foundations

Last time

- Intro to ACID transactions
- Focus on Isolation
  - Every transaction has the illusion of having the DB to itself
- Isolation anomalies
  - bad things that can happen due to interleaving

Transaction 1

```
UPDATE accounts
SET amount= amount - 100
WHERE name = 'Alice';

UPDATE accounts
SET amount= amount + 100
WHERE name = 'Bob';
```

Transaction 2

```
UPDATE accounts
SET amount= amount*1. 1
WHERE name = 'Alice';

UPDATE accounts
SET amount= amount*1.1
WHERE name = 'Bob';
```

Bad interleaving #1

Transaction 1	Transaction 2
UPDATE accounts SET amount= amount- 100 WHERE name = 'Alice';	
	UPDATE accounts SET amount= amount*1.1 WHERE name = 'Alice';
	UPDATE accounts SET amount= amount*1.1 WHERE name = 'Bob';
UPDATE accounts SET amount= amount+ 100 WHERE name = 'Bob';	

Bad interleaving #1

- Let's introduce some notation to make it easier to write these down
- Transactions work on objects A, B, C
- Transactions read (R) and Write (W) objects

```
UPDATE accounts
SET amount= amount- 100
WHERE name = 'Alice';
```

- This becomes R1(A) W1(A)
  - A = tuple with Alice's info
  - R because of reference to amount on RHS
  - W because we change amount
  - 1 because we're in transaction 1

Bad interleaving #1

Transaction 1	Transaction 2
UPDATE accounts SET amount= amount- 100 WHERE name = 'Alice';	
	UPDATE accounts SET amount= amount*1.1 WHERE name = 'Alice';
	UPDATE accounts SET amount= amount*1.1 WHERE name = 'Bob';
UPDATE accounts SET amount= amount+ 100 WHERE name = 'Bob';	

Bad interleaving #1

Transaction 1	Transaction 2
R1(A) W1(A)	
	R2(A) W2(A)
	R2(B) W2(B)
R1(B) W1(B)	

- More shorthand: R1(A) W1(A) becomes RW1(A)

Reading uncommitted data

- What is the essence of the problem?
  - RW1(A) RW2(A) RW2(B) RW1(B)
  - Transaction 2 has read the value of A written by Transaction 1 before Transaction 1 committed
  - Who knows whether Transaction 1 was done modifying A?
  - In this case it was, but it needed to make a corresponding change to B which 2 didn't see
- Another variant – read from someone that later aborts
  - RW1(A) RW2(A) Abort1 RW2(B) Commit2

Dirty read

- Reading a value modified by an uncommitted transaction
  - Not **guaranteed** to cause a problem in all cases, but
  - Particularly bad if the transaction you read from aborts after your read
  - But you see it can be bad even with no aborts

Dirty read

- Why bad:
  - Don’t know if transaction is done changing the value
  - Or other values that are related to it by an invariant
  - If transaction aborts, we are really in trouble.

Unrepeatable read

- The second isolation anomaly
- Even if we never read uncommitted data, can still have a different problem
- R1(A) W2(A) C2 R1(A)
- Read the same data item twice, get different answer
  - Real world example: first read checks if enough inventory available, second read is part of decrement operation on inventory

Lost updates

- The third isolation anomaly
- Fundamentally related to writes (not reads)

Transaction 3 – W(A) W(B)

```
UPDATE accounts
SET amount =  100
WHERE name = 'Alice';

UPDATE accounts
SET amount =  100
WHERE name = 'Bob';
```

Transaction 4 – W(A) W(B)

```
UPDATE accounts
SET amount = 200
WHERE name = 'Alice';

UPDATE accounts
SET amount = 200
WHERE name = 'Bob';
```

Lost updates

- Suppose transactions interleave as follows:
- W3(A) W4(A) W4(B) W3(B)
  - At the end Alice has $200 and Bob has $100, so the amounts are not equal…
  - If we had equality as a consistency constraint, it has just been violated
- Although no-one read an inconsistent state, somehow some writes got mixed up giving an inconsistent DB at the end

The three isolation anomalies

- Dirty read
  - Read data someone else is not finished with
  - RW1(A) RW2(A) RW2(B) RW1(B)
- Unrepeatable read
  - Data is changing under you
  - R1(A) W2(A) C2 R1(A)
- Lost Update
  - Writes are otherwise "correct" but destroy each other in a bad way
  - W3(A) W4(A) W4(B) W3(B)

Avoiding Anomalies

- How do we know what all the possible anomalies are?
  - So that we can design an algorithm to prevent them…
- Are some of them more serious than others?

A more principled approach

- Definition: transaction schedule
  - A sequence of operations performed by a set of transactions on the DB
  - e.g. W1(A) W2(A) W2(B) W1(B) Commit2 Commit1
- A transaction schedule is a concrete interleaving
  - We can identify “good” and “bad” schedules
  - E.g. if exhibits one of our anomalies, bad!!
  - Once we know what's "good" and "bad", we can start understanding how to avoid "bad" schedules

Serializability

- Intuition: suppose transactions in the schedule had executed serially
  - e.g. W1(A) W1(B) C1 W2(A) W2(B) C2
- Then nothing bad could have happened!
  - Alice and Bob banking example

Bad interleaving

Transaction 1	Transaction 2
UPDATE accounts SET amount= amount- 100 WHERE name = 'Alice';	
	UPDATE accounts SET amount= amount*1.1 WHERE name = 'Alice';
	UPDATE accounts SET amount= amount*1.1 WHERE name = 'Bob';
UPDATE accounts SET amount= amount+ 100 WHERE name = 'Bob';	

Serializability

- Idea: compare our schedule of interest to a serial schedule with the same transactions
  - We want to "simulate serial execution"

Serializability

- First requirement: a good schedule is one that produce the same final DB as some serial schedule (with same transactions)
  - if doesn't produce same final DB as any serial schedule, this is pretty iffy!
    - How can we have any guarantee the DB is consistent in this case?

Final-state serializability

- A schedule S is final-state serializable if there is some serial schedule S' involving the same transactions such that the final DB is the same after executing S and S'
  - No matter what DB you start with
- Example: W1(A) W2(A) W1(B) W2(B) C1 C2

Do we need something stronger?

- R1(A) W2(A) R1(A) C1 C2
- Unrepeatable read…
- But final DB produced is the same as after serial execution where 2 executes before 1!
- Apparently requiring same final DB may not be enough!
  - Reads matter too
  - T1 should see the same value of A if it reads it twice (without making any changes of its own)

View Equivalence

- Intuition: If transaction i sees a value written by transaction j in schedule S1 (it is "tainted" or "contaminated" by j), it should also see value written by transaction j in Schedule S2
- There is a dependency (an ordering) between j and i
- Two schedules are "similar" if they have all the same dependencies between transactions

View Equivalence

- Schedules S1 and S2 are view equivalent if:
  - If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
  - If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
  - If Ti writes final value of A in S1, then Ti also writes final value of A in S2

T1:	R(A)		W(A)	
T2:		W(A)		
T3:				W(A)

T1:	R(A),W(A)		
T2:		W(A)	
T3:			W(A)

View Serializability

- A schedule S is view serializable if it is view-equivalent to some serial schedule
- Let's see why our "offending" schedule is not view serializable...
- R1(A) W2(A) R1(A) C1 C2

So what now?

- Want to allow only view-serializable schedules (interleavings) in our system
- Will need some algorithm to achieve this
- Useful to define a third kind of serializability
  - conflict-serializability
- It is NP-complete to test whether a schedule is view-serializable
  - But conflict-serializability very easy to check efficiently